

Generic Programming of Reusable, High Performance Container Types using Automatic Type Hierarchy Inference and Bidirectional Antichain Typing

Wouter Kuijper

W.Kuijper@utwente.nl
University of Twente

Michael Weber

M.Weber@utwente.nl
University of Twente

Abstract

We introduce a new compile-time notion of type subsumption based on type simulation. We show how to apply this static subsumption relation to support a more intuitive, object oriented approach to generic programming of reusable, high performance container types. As a first step towards an efficient implementation of the resulting type system in a compiler we present a novel algorithm for bidirectional type inference over arbitrary syntax graphs. The algorithm uses the new static type subsumption relation to compress the data that has to be stored for each node in the typeflow graph. During typeflow analysis this means that the set of types for a given node can be symbolically represented using antichains instead of using bitvectors or some other explicit set representation. This results in a typing algorithm that is both flexible and precise and shows good performance on representative instances.

1. Introduction

Besides their useful role in enforcing partial correctness, types play an important role in program synthesis. Not only does a well designed type system prevent the programmer from specifying certain unsafe operations, types also serve to disambiguate programs. This is the case for languages that support some form of *function overloading* where argument types and return type determine the particular function implementation that is invoked.

Many among the most popular programming languages to date are dynamic languages. This means that, to a more or lesser degree, function overloading is dealt with at run-time. This is not surprising as it is generally more programmer-friendly than generic programming with compile-time type-substitution. The latter technique constitutes the only *truly* static alternative that is available today for writing high performance, reusable container types. The distinguishing feature of the generic programming technique is that it makes use of lexical substitution of types through *type parameters*.

Using *templates* and *generic types* it becomes possible to completely eliminate all overhead due to dynamic type checks because all information about types can be fixed at compile-time. As such, the aforementioned programming constructs are used mainly for

performance critical applications where the overhead of dynamic checks to resolve overloading cannot be sustained.

However, obtaining this performance comes at a price. Programming generic code can be difficult, labor intensive (due to the very explicit way type parameters must be passed through every syntactical construct) and counter intuitive. Quoting Stroustrup [17] on C++ templates:

As far as the C++ language rules are concerned, there is no relationship between two classes generated from a single class template. For example:

```
class Shape { /*...*/ };
class Circle:public Shape { /*...*/ };
```

Given these declarations, people sometimes try to treat a `set<Circle*>` as a `set<Shape*>`. This is a serious logical error based on a flawed argument: “A `Circle` is a `Shape`, so a set of `Circles` is also a set of `Shapes`; [...]”

Bjarne Stroustrup
(*The C++ Programming Language*)

For most programmers this is counter intuitive: if a `Circle` is a `Shape` then intuition tells us that a `Set` of `Circles` must be a `Set` of `Shapes`. In Java Generics, the modern descendant of the C++ template system, this particular situation has not improved. Quoting Bracha [3] on Java Generics:

In general, if `Foo` is a subtype (subclass or subinterface) of `Bar`, and `G` is some generic type declaration, it is not the case that `G<Foo>` is a subtype of `G<Bar>`. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

Gilad Bracha
(*Generics in the Java Programming Language*)

This counter intuitive trait in current generic programming approaches is caused by the fact that type subsumption of the underlying languages is essentially a dynamic notion that is designed to be resolved at *run-time*. So we see that there is a real need for a programmer-friendly way of dealing with static type hierarchy designed to be resolved at *compile-time*.

As a solution to this problem we propose a new notion of *static type subsumption*. This new subsumption relation can be used in conjunction with the currently prevalent notion of *dynamic type subsumption*. By making this distinction more clearly we are treating the static type subsumption relation as a first class citizen. In particular this means that it becomes possible to use generic programming *in an object oriented style*, i.e.: if `Shape` statically subsumes `Circle` this will imply that `Set<Shape>` stati-

cally subsumes `Set<Circle>`, and `List<Set<Shape>>` statically subsumes `List<Set<Circle>>`, etc.¹

The dynamic type subsumption relation will be, in general, a subset of the static type subsumption relation where additional alignment constraints must be met. In this paper we will focus exclusively on computing and exploiting the static type subsumption relation.

1.1 Contribution and Structure of the Paper

The contribution of this paper is twofold. In Section 4 we present a type system that combines structural typing, function overloading and static type strengthening as an intuitive, object oriented alternative to existing generic programming approaches. In Section 5 we present a new bidirectional antichain typing algorithm that allows a practical, efficient implementation of this new type system.

The paper is structured as follows. In Section 2 we discuss related work. In Section 3 we give a motivating example which will also serve as a running example for illustrating the definitions in the following sections. In Section 4 we give a formalization of the type hierarchy as a simulation relation and we explain how to infer type hierarchy from surface level declarations and definitions. In Section 5 we show how to subsequently use the inferred type hierarchy to efficiently type programs. In Section 6 we give some perspectives on our current results and future work.

2. Related Work

Types were originally invented as simple names for sets of values that form part of a programming language [12]. As programming languages grew more sophisticated in keeping up with the complexity of the problems they were employed to solve, types evolved into more than simple sets of values and became an object of study in and of itself.

It was observed that types were instrumental in enforcing all kinds of safety constraints thus obtaining a form of partial correctness [4, 10].

It was found independently by Hindley [7] and Milner [11] that polymorphic types were useful for structuring programs. Since the early contributions of Hindley and Milner many alternatives and extensions to their approach have been suggested in the literature [1, 2, 5, 14–16].

More recently other uses of types are being explored that enrich type systems in various ways with constraints and qualifiers that can express certain invariants and in this way further the role of types in writing safe and correct programs [6, 13].

Another line of work proposes to improve the flexibility of typing programs by treating the typing problem using a form of dataflow analysis [8, 9, 18]. The work in this paper is related to the dataflow approach as we also exploit the structure of the syntax graph to explicitly guide the typing process. However we use an inherently bidirectional antichain algorithm that does not immediately fit within the dataflow framework.

Antichains have recently received notable attention for their potential use as a symbolic representation for upward or downward closed sets. As such, with antichains we can solve many problems in formal language theory much more efficiently than classical algorithms that, typically, require a subset construction [19].

3. A Motivating Example

The MOOT (modular object oriented template) programming layer is a thin experimental programming layer over a small subset of

C++. It adds static function overloading and static type strengthening to the basic C-like subset which forms the core of the C++ language.

Informally we say that type A *statically subsumes* type B (or type B is *stronger* than type A) iff all the relevant operations that can be compiled for type A can also be compiled for type B. In effect this is a form of simulation relation between type A and B with respect to the operations that objects of type A and B support. We will discuss this formally in Section 4.

As *relevant* operations (relevant to the type subsumption ordering) we take all the built in operations, structural field select operations, pointer and array dereference and the “operation” of being passed to or returned from a function that is part of some formal protocol definition.

As an example we consider the following MOOT definition of a protocol for iterating over a collection of values:

```
protocoltype Iterable;

protocoltype Iterator;

void FIRST( Iterable c, Iterator &e );

bool DONE( Iterable c, Iterator e );

void NEXT( Iterable c, Iterator &e );

any DATA( Iterable c, Iterator e );
```

As a first example we instantiate this protocol for counting up to some integer value:

```
void FIRST( int c, int &e ) { e = 1; }

bool DONE( int c, int e ) { return e > c; }

void NEXT( int c, int &e ) { e++; }

int DATA( int c, int e ) { return e; }
```

Now the following is a valid application:

```
int x = 5; int y;
for ( FIRST(x,y); !DONE(x,y); NEXT(x,y) ) {
    printf( "%d; ", DATA(x,y) );
}
```

Which prints: 1; 2; 3; 4; 5; Now let us instantiate this protocol for integer intervals:

```
struct _Ival {
    int min;
    int max;
};

typedef struct _Ival Ival;

void FIRST( Ival+ c, int &e ) { e = c.min; }

bool DONE( Ival+ c, int e ) { return e > c.max; }

void NEXT( Ival+ c, int &e ) { e++; }

int DATA( Ival+ c, int e ) { return e; }
```

Here the plus + type qualifier signals that the declared type is *compile-time strengthenable*, meaning the functions may also be invoked with arguments of a stronger type taken from the downward closed set of types under `Ival+` in the subsumption hierarchy of types as shown in Figure 1. We say the type may be *strengthened* if this is necessary to get a type correct program.

¹ Given that `List` and `Set` are generic container types that support these type substitutions.

Now that we have two different instantiations of the protocol it is useful to have a generic function template for printing `Iterable` collections:

```
void print( Iterable+ c ) {
    Iterator+ e;
    for ( FIRST(c,e); !DONE(c,e); NEXT(c,e) ) {
        print( DATA(c,e), " " );
    }
}
```

```
void print( any+ a, any+ b ) {
    print(a); print(b);
}
```

```
void print( int+ i ) {
    printf( "%d", i );
}
```

```
void print( char+ *s ) {
    printf( "%s", s );
}
```

We use `any` as a special type that subsumes all types (in effect it denotes the top element of the type lattice). The following is then a valid application:

```
Ival i; i.min = 11; i.max = 15;
print(i, "\n");
```

Which prints: 11; 12; 13; 14; 15; To give a slightly more interesting example, we want to subclass the integer interval with a directed interval that includes an extra field for iterating over the interval from either side using a given increment. This can be done as follows:

```
struct _DirIval {
    int min;
    int max;
    int delta;
};
```

```
typedef struct _DirIval DirIval;
```

```
void FIRST( DirIval+ c, int &e ) {
    if ( c.delta > 0 ) e = c.min; else e = c.max;
}
```

```
bool DONE( DirIval+ c, int e ) {
    return (c.delta > 0 ? e > c.max : e < c.min);
}
```

```
void NEXT( DirIval+ c, int &e ) {
    e += c.delta;
}
```

We may leave out the definition of `DATA` because it carries over from the old definition with `Ival`. The following is then a valid application:

```
DirIval d; d.min = 11; d.max = 15; d.delta = -2;
print(d, "\n");
```

Which prints: 15; 13; 11;.

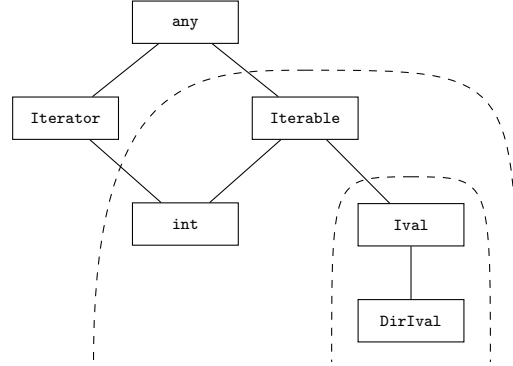


Figure 1. Inferred type subsumption order for some of the types from the Example in Section 3, the dashed areas show the downward closed sets of types denoted with `Iterable+` and `Ival+` respectively.

3.1 Translating Type Strengthening to C++

Space constraints limit us from describing the MOOT programming layer in too much detail. However, since in the example we are using only a single new syntactical construct (the `+` strengthability type qualifier) layered on top of the basic C language we will explain the semantics of this construct by showing how it compiles down to C++. For example, the definitions of the protocol functions for the type `DirIval` compile down to:

```
void FIRST_DirIval_int( DirIval c, int &e ) {
    if ( c.delta > 0 ) e = c.min; else e = c.max;
}

bool DONE_DirIval_int( DirIval c, int e ) {
    return (c.delta > 0 ? e > c.max : e < c.min);
}

void NEXT_DirIval_int( DirIval c, int &e ) {
    e += c.delta;
}

int DATA_DirIval_int( DirIval c, int e ) {
    return e;
}
```

Note that the last function has been inherited from `Ival`. Also note that the type information has been pushed onto the identifier names to make them unique. The print function for the same type compiles down to:

```
void print_DirIval( DirIval c ) {
    int e;
    for ( FIRST_DirIval_int(c,e);
          !DONE_DirIval_int(c,e);
          NEXT_DirIval_int(c,e) ) {
        print_int( DATA_DirIval_int(c,e) );
    }
}
```

As can be seen all the function overloading has been completely and statically resolved by the MOOT layer and the program can be readily compiled by any C++ compiler.

Resolving calls to overloaded functions and strengthening the strengthenable declarations to their proper types first requires us to infer the type hierarchy in the form of the type subsumption relation as shown in Figure 1. We come back to this in Section 4.

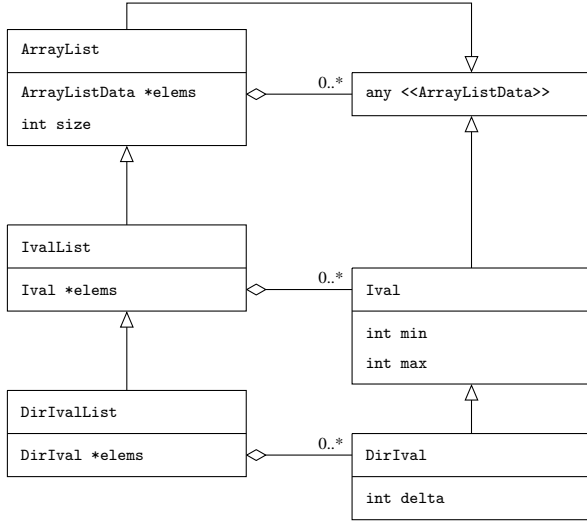


Figure 2. Class-diagram for some of the types in the example.

3.2 Container Types

In the previous section we showed how the MOOT layer is capable of resolving function overloading and automatic type strengthening at compile-time which are important prerequisites for an object oriented generic programming layer. The second crucial ingredient for any generic programming language is the instantiation of types from generic container types.

Perhaps surprisingly it is not at all difficult to support the instantiation of types from generic container types. This is mainly a matter of lexical substitution of types which just requires the appropriate syntax. The difficult part is to keep track of relations between the resulting types *after* these substitutions have been carried out by the compiler.

Note in the previous example that we do not require the programmer to make any explicit declaration of type subsumption. This is a very important feature of the programming layer. It now becomes possible to build container types whilst maintaining the static type subsumption relation. For example, in MOOT a simple, generic arraylist structure might be defined as follows:

```
parametertype ArrayListData;

struct _ArrayList {
    ArrayListData *elems;
    int size;
};

typedef struct _ArrayList ArrayList;
```

This generic list datastructure can now be instantiated using parameterized typedef declarations:

```
typedef
    ArrayList<Ival ArrayListData>
    IvalList;

typedef
    ArrayList<DirIval ArrayListData>
    DirIvalList;
```

In MOOT it now follows automatically that `IvalList` statically subsumes `DirIvalList` without any additional help from the pro-

grammer. So, in particular, it is possible to strengthen any declaration of `IvalList+` to `DirIvalList+`.

The semantics of the `<...>` parameterized typedef construct can be defined purely lexically: the type substitutions occurring between the angled brackets are carried out on the original type definition (and, transitively, on any struct or typedef definition on which it depends). For the example, the end result compiles down to normal type declarations as follows:

```
struct _IvalList {
    Ival *elems;
    int size;
};

typedef struct _IvalList IvalList;

struct _DirIvalList {
    DirIval *elems;
    int size;
};

typedef struct _DirIvalList DirIvalList;
```

As can be seen the result depends on naming conventions: the original type name is substituted with the new type name.

Note that, without a structural type system, it would be impossible to conveniently maintain the proper type subsumption relation *and*, at the same time, allow such a powerful lexical construct like the `<...>` typedef parameter construct.

3.3 Parameter Types with Protocol Assumptions

Substitution of parameter types for arbitrary types is a common design pattern used in generic programming due to the fact that container types are usually intended to work for *any* type. No assumptions are made on the internal structure of the underlying data or on the operations available for the underlying data.

However, opaque parameter types without any assumptions placed on them are not always sufficient. There are certain cases where we would like to provide specialized, or optimized functionality for data that satisfies certain additional assumptions. One example would be a datastructure for an ordered list that relies on a comparison function being available over the underlying data. In MOOT it possible to formalize this additional assumption using protocols. As an example consider the following refinement of our simple array list:

```
protocoltype Comparable;

bool LTE( Comparable x, Comparable y );

parametertype SortedArrayListData : Comparable;

typedef
    ArrayList<SortedArrayListData ArrayListData>
    SortedArrayList;
```

Now the new `SortedArrayList` parameter type inherits the `LTE` (less-than-or-equal) protocol operation from protocol type `Comparable`. We use this new parameter type to define a derived container type `SortedArrayList` that should keep the elements in the arraylist sorted. We will not work this example out further in this paper. However if we were to define the implementation of `SortedArrayList` we would do so in terms of the parameter type `SortedArrayListData`. In each function that we would write as part of this implementation we could then safely assume the existence of a suitable function `LTE` that implements the underlying ordering.

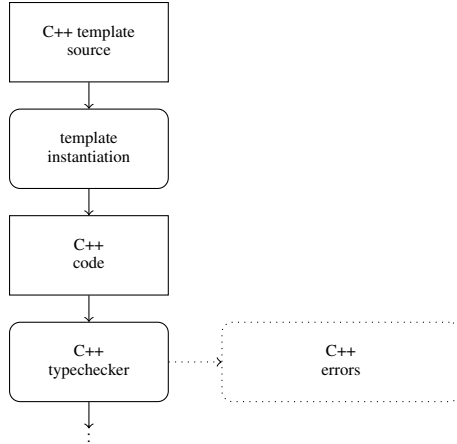


Figure 3. Reporting of type errors for C++ templates.

If the user would try to instantiate our new, sorted datatype without defining a suitable LTE operation the compiler would detect this by checking whether the instantiated type is subsumed by the original declaration, i.e. if the user would now declare:

```
typedef
  SortedArrayList<Ival SortedArrayListData>
  SortedIvalList;
```

The compiler would give the following error message:

```
SortedArrayListData does not subsume Ival
missing: LTE( Ival), ... );
```

As shown in Figure 3 the traditional approach to template languages does not allow such checks to be performed before the actual type-checking phase is entered. In Figure 4 we show how this situation is improved in MOOT.

3.4 Object Orientation and Type Hierarchy

Because type hierarchy in MOOT is inferred automatically, much of the syntax that is traditionally present in object oriented languages is missing in MOOT². Nevertheless, the primitives we discussed so far offer us enough freedom to build our own object systems conveniently. As such MOOT offers us the ability to use generic programming in an object oriented style.

To illustrate this, Figure 2 shows some of the types introduced in the running example. The diagram shows the proper type subsumption relation as the inheritance relation and the type relations induced by the struct fields as aggregations. The result is a “class-diagram” of our types.

Note that, in order to keep the class-diagram compact, we left out the operations (FIRST, NEXT, DONE, DATA, etc.). These operations that take values of the various types as their first arguments would typically be included in such a class-diagram as *methods*. In this context it is important to note that, in order to infer this type hierarchy, we need to deal with a potential form of circularity that arises when we adopt the proposed structural definition of type subsumption together with the notion of type strengthening for function arguments.

In fact this potential circularity *is* present in the example. In the one direction: the reason that `DirIval` is subsumed by `Ival`

²In this paper we do not discuss all the syntax that MOOT offers. In particular it is possible to explicitly inherit a `struct` type from another `struct` type. But in contrast to other object oriented languages this is syntactic sugar rather than a primitive construct.

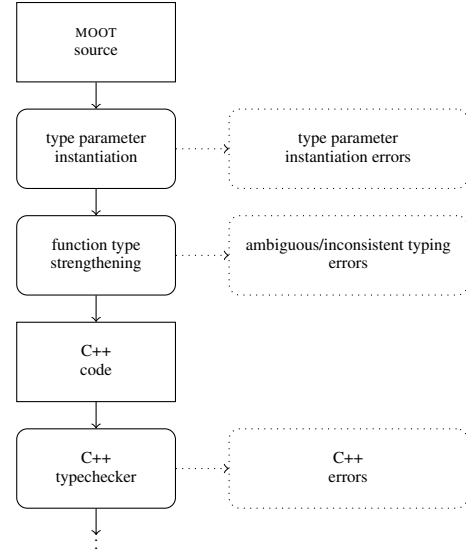


Figure 4. Early warning of type errors in the case of MOOT.

is that all the field select operations (`.min`, `.max`), and all the protocol operations (FIRST, NEXT, DONE, DATA) which are available for `Ival` are also defined for `DirIval`. In the other direction: the DATA operation is defined for `DirIval` because it is inherited from `Ival` and this only works precisely because `DirIval` is subsumed by `Ival`, which entails `Ival+` declarations may be *strengthened* to `DirIval+` declarations.

In general the potential circularity of reasoning can be broken by defining the type subsumption relation as the largest possible type simulation relation that is mutually consistent with the rules for `protocoltype` and `struct` subsumption. We will come back to this in Section 4.

3.5 Calling Functions with Strengthenable Arguments

So far we have discussed how parameter types allow us to parameterize and instantiate generic type declarations. In order to implement these types we need to define functions over them. In MOOT function definitions are never instantiated through the `<...>` angled bracket notation, this notation is reserved for type declarations. For function definitions we rely solely on *type strengthening*. For this it is important to understand how a call to a function with strengthenable arguments is resolved, i.e.: which of the various overloaded function bodies is actually invoked? As an example we take the previously defined `print/1` function.

In the example we overloaded the `print/1` function several times. Now we did not define `print/1` as part of a formal protocol. As such it does not influence the type subsumption relation. However, in the other direction, the type subsumption relation *does* influence how calls to `print/1` get resolved.

In particular we gave a definition with signature `print(int+)` for simple integers, and we gave a second definition with signature `print(Iterable+)` for `Iterable` values. At the same time we implemented the `Iterable` protocol for simple integers. This means that we need a principle on which to resolve a call like: `print(3)`: do we map it to the former or to the latter function definition?

We answer this following the usual semantics which means we resolve to the *strongest possible* signature. In this case the signature `print(int+)` is stronger than the signature `print(Iterable+)` because `int` is `Iterable` but not the other way around. In MOOT we provide syntax to fine tune the matching of the function

on one or more arguments by weakening the signature against which the function is matched. For the example we might write `print([~Iterable]3)`. Which prints: 1; 2; 3;

3.6 Multiple Strengthenable Arguments

The only remaining issue concerning the semantics of the new strengthenable type qualifier arises when there is *more than one* strengthenable formal parameter to some defined function.

In order to understand what would be the right call matching semantics for functions with multiple strengthenable arguments it is good to look at some pathological cases and see how we should best deal with these cases, that is: providing minimal confusion to the programmer. First, consider the situation where we would provide the following two function definitions with the same name and arity:

```
DirIval+ intersect( DirIval+ i1, Ival+ i2 )      (1)
```

```
DirIval+ intersect( Ival+ i1, DirIval+ i2 )      (2)
```

We say these two function signatures are *incomparable*, because the first definition is stronger in the first argument type, whereas the second definition is stronger in the second argument type. When we consider which of the functions to call in an application like the following:

```
DirIval d1, d2, d3;
...
d1 = intersect( d2, d3 );
```

It follows we must either pick one of the two function definitions or we must reject the program with a typing error. The first option is problematic because it introduces an element of arbitrariness into the semantics. Therefore, in this case, we prefer the second option.

3.7 Singleton Antichain Semantics

Now consider what should happen if, in addition to function definitions (1) and (2) we would add a third function definition:

```
Ival+ intersect( Ival+ i1, Ival+ i2 )      (3)
```

For our example, with respect to the call matching semantics for `intersect(d2, d3)`, we have a third option to consider namely to invoke function definition (3). Even though it is strictly *weaker* than function definitions (1) and (2), it at least lacks the element of arbitrariness. To see this just note that the antichain of incomparable functions with the same name and arity as function definition (3) contains only function definition (3) itself. As such, we will refer to this tentative semantics as the *singleton antichain semantics*.

The singleton antichain semantics does not suffer from the arbitrariness we discussed earlier. However, there is another reason to reject this semantics. In practice what happens when programmers use function overloading is that definitions are grouped, conceptually, into *classes* which often also end up being defined in different source files (*modules*). For our example this might mean that function definitions (1), (2) and (3) occur far removed from each other.

Now consider a programmer who completes function definition (3) *first* and subsequently goes on and overloads this definition with function definition (2). The program might compile and work for a while until, at some point, somebody decides to add function definition (1) without paying attention to the existence of function definition (2). Given the singleton antichain semantics this would mean that the new function definition (1) would be silently ignored because of the existence of function definition (2), and, vice versa, the existing function definition (2) would now also be silently ignored because of the existence of the new function definition (1). More seriously even, in all the cases where we used to invoke function definition (2) we would then *go back* to the invocation of the

older function definition (3). So we see that adding a new function under such a call matching semantics may have unexpected, non-local effects.

3.8 Strongest Call Semantics

The latter example shows that the singleton antichain call matching semantics would also be problematic. For this reason we propose the *strongest call semantics*. Under this call matching semantics a function definition is only invoked iff *all* the formal parameters in the signature, pointwise, are the strongest possible match to the types of the corresponding actual parameters among all of the defined function signatures (of the same name and arity). Under this semantics the call `intersect(d2, d3)` remains untypeable also when function definition (3) is added. To fix this situation the programmer may always introduce a fourth function definition:

```
DirIval+ intersect( DirIval+ i1, DirIval+ i2 )  (4)
```

To avoid the non-local effects that we mentioned earlier we may warn the user when two incomparable function definitions (with the same name and arity) occur in different source files. Together with the strongest call semantics this enforces a reasonable level of modularity. In Section 5 we will show how to formalize and enforce the strongest call semantics for function call expressions.

4. Inferring Type Hierarchy

In Section 3 we already briefly remarked that we view type subsumption as a form of simulation relation, in this section we will discuss this formally. The analysis in this section is based on the assumption that we can obtain a finite set of relevant types from the source code of the program. We will assume that the user will provide all the relevant types³. If it turns out that the program cannot be typed because the user forgot to provide a type this will be flagged with a clear error message. However, the typing procedure will never introduce new types outside the finite set of relevant types. In this way we ensure termination. Below we give the basic definition of type subsumption as a simulation relation.

Definition 1 (Type Simulation and Subsumption) Let T be a finite set of *relevant types*, and let $\{R_\sigma\}_{\sigma \in \Sigma}$ be a finite, indexed set of *type relations* over T , i.e. for all $\sigma \in \Sigma$ it holds $R_\sigma \subseteq T \times T$. For all $\sigma \in \Sigma$ let $R_\sigma^\dagger \subseteq R_\sigma$ be a selected subset of the type relation, we say the tuples in R_σ^\dagger are *strengthenable*. For a given candidate subsumption relation $\mathcal{R}_\preceq \subseteq T \times T$ and any two types $t, t' \in T$ we define $t \simeq_{\mathcal{R}_\preceq}^\sigma t'$ iff for all $u' \in T$ such that $t' R_\sigma u'$ there exists some $u \in T$ such that $u R_\preceq u'$ and $t R_\sigma u$ or $t(R_\preceq; R_\sigma^\dagger)u$ where $;$ denotes relation composition. We say some candidate subsumption relation $\mathcal{R}_\preceq \subseteq T \times T$ is a σ -simulation relation iff for all $(t, t') \in \mathcal{R}_\preceq$ it holds that $t \simeq_{\mathcal{R}_\preceq}^\sigma t'$. We say some candidate subsumption relation $\mathcal{R}_\preceq \subseteq T \times T$ is *valid* iff for all $\sigma \in \Sigma$ it holds \mathcal{R}_\preceq is a σ -simulation. To compute the greatest valid subsumption relation based on this requirement we may define the following fixed point operation:

$$F(\mathcal{R}_\preceq) = \bigcap_{\sigma \in \Sigma} \{(t, t') \in \mathcal{R}_\preceq \mid t \simeq_{\mathcal{R}_\preceq}^\sigma t'\}$$

Since $F(\cdot)$ is monotone it always has a greatest fixed point. Let $\mathcal{R}_\preceq^{\text{init}}$ be some pre-order that forms an initial, syntactical overapproximation of the desired type subsumption relation. We now define \preceq as the *largest valid subsumption relation* contained in $\mathcal{R}_\preceq^{\text{init}}$.

³ In practice (for MOOT) we do saturate the set of relevant types with 1-deep pointer types to avoid overly pedantic errors, further note that instantiating a parameterized container type may introduce a significant amount of types without any work for the user, given that any type that the container type transitively depends on is also instantiated automatically.

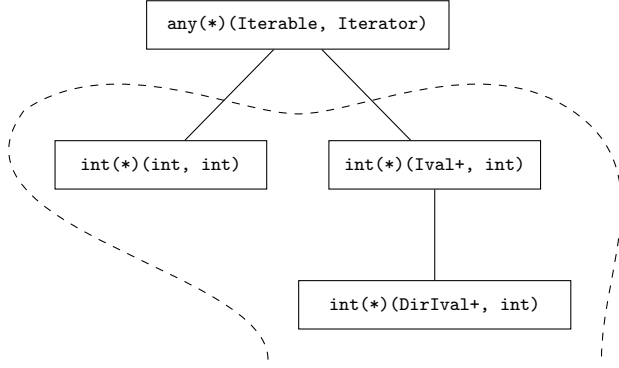


Figure 5. Inferred type subsumption order for some of the function types of the Example in Section 3, the dashed area shows the downward closed set of types denoted by the antichain of function types: $\{\text{int}^*(\text{int}, \text{int}), \text{int}^*(\text{Ival}^+, \text{int})\}$.

When we speak about the *simulation graph* we mean the graph over T that includes \preceq and all the other type relations $\{R_\sigma\}_{\sigma \in \Sigma}$. \square

This definition is generally applicable and still allows a great deal of freedom in the actual details of the type system. We will now sketch a number of examples where we make use of the definition of type subsumption as a simulation relation in the MOOT type system.

4.1 The Role of Type Hierarchy in MOOT

The MOOT programming layer is specifically meant to deal with *static type subsumption*. In a performance oriented language like C++ this is an important concept: if everything about types is known at compile-time this means we can avoid introducing run-time checks, avoid tagging data-structures with type-identifiers, and employ the C++ compiler to optimize many operations.

In this context it is important to note that the structural type system is introduced mainly to free the programmer from the burden of having to deal with an overly rigid type system, and to do so with minimal impact on code readability and efficiency. As such the type subsumption relation that we will infer using the techniques described here should *not* be thought of as the final safeguard that stands between the program and its execution. Instead, the type subsumption relation serves mainly as an aid to structure the set of types which benefits both the user (by allowing concise and understandable code) as well as the typing procedure (by allowing the partial order structure to be exploited for efficiency).

The final result will be compiled down to statically typed C++ and any missing or illegal operations that are not caught by the MOOT type system will be rejected at the next stage of compilation, since the C++ type system is more strict than the MOOT type system. The syntactical layer is thin enough to allow the user to understand how an error message from the C++ compiler relates back to the original code. This is especially true because line-numbering, statements, expressions and control flow are fully preserved. Also, because we do not present the user with a raw trace of the type algebraic expressions (as is done in a template language like STL) the error messages are in fact more understandable in the case of MOOT.

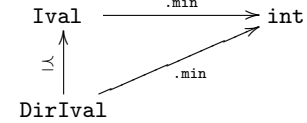
4.2 Simulation of Structural Types

As a first example we will consider the structural types `Ival` and `DirIval` as they were defined in Section 3, and the `.min` field

selection operation. We will formalize this as a type relation $R_{\text{.min}}$ such that $(t, u) \in R_{\text{.min}}$ iff t is a `struct` type that contains a field `min` of type u . For the example this becomes:

$$R_{\text{.min}} = \{\text{Ival} \mapsto \text{int}, \text{DirIval} \mapsto \text{int}\}$$

Now in order to see that `DirIval` simulates `Ival` with respect to this type relation we must check whether the result after applying `.min` on `DirIval` still simulates the result after applying the same operation on `Ival`. This simulation condition can be summarized in the following subdiagram of the simulation graph:



In this case the simulation requirement is fulfilled. For simple direct type relations like selecting a field in a `struct` it is quite straightforward to check the simulation requirement. This approach also works for recursive `struct` types.

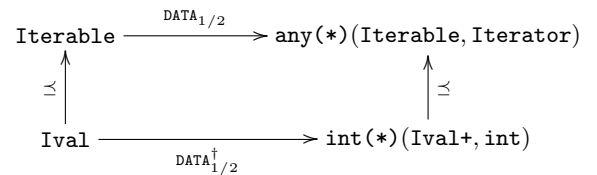
4.3 Simulation of Function Types

Treating function types is somewhat more involved. We give as an example the `DATA` operation. On the level of types we formalize the type relation $R_{\text{DATA}_{1/2}}$ meaning "possible signature based on first argument type to the binary operation `DATA`". More specifically, we place a pair (t, f) of a type and a function type in the argument-signature relation $(t, f) \in R_{\text{DATA}_{1/2}}$ iff there exists a *definition* of binary operation `DATA` with *declared signature* f that contains t as the first argument, moreover we put $(t, f) \in R_{\text{DATA}_{1/2}}^\dagger$ if, in addition, t is marked as a strengthenable argument in f . For the example this becomes:

$$\begin{aligned} R_{\text{DATA}_{1/2}} &= \\ &\{ \text{Iterable} \mapsto \text{any}^*(\text{Iterable}, \text{Iterator}), \\ &\quad \text{int} \mapsto \text{int}^*(\text{int}, \text{int}), \\ &\quad \text{Ival} \mapsto \text{int}^*(\text{Ival}^+, \text{int}) \} \\ R_{\text{DATA}_{1/2}}^\dagger &= \\ &\{ \text{Ival} \mapsto \text{int}^*(\text{Ival}^+, \text{int}) \} \end{aligned}$$

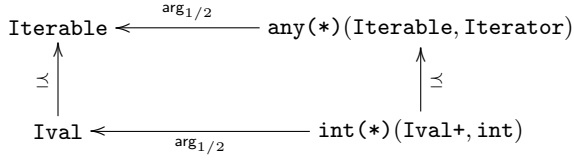
Here $\text{any}^*(\text{Iterable}, \text{Iterator})$ is the C type signature notation for a function that takes a value of `Iterable` type as its first argument, a value of `Iterator` type as its second argument and returns a value of `any` type.

The simulation requirement between `Ival` and the `Iterable` protocol, on the first argument of the binary `DATA` operation, can then be summarized in the following subdiagram of the simulation graph:

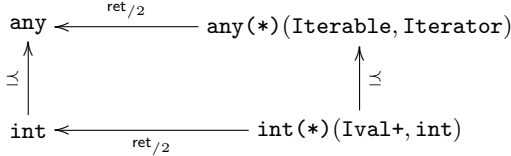


We see that this simulation requirement depends on the simulation between two function types. So we need to be explicit about when two function types, like the ones shown in the diagram, are in the subsumption relation. To do this within the current framework we introduce new type relations $\text{arg}_{i/j}$ for selecting the i -th argument

type from a j -ary function type:



So in effect we see that the simulation requirement runs in both directions in this case. For the second argument this works likewise. For the return type we introduce a type relation $\text{ret}_{/j}$ for selecting the return type from a j -ary function type:



More precisely; a function type A subsumes a function type B iff they are of the same arity and all the arguments of A subsume the corresponding arguments of B and the return type of A subsumes the return type of B and at least all the places in which A is strengthenable are also strengthenable in B (the latter condition is enforced through $\mathcal{R}_{\preceq}^{\text{init}}$).

As can be seen the subsumption condition on function types is covariant between arguments and return type. In this context it is good to recall that we are considering only *static type subsumption*. For *dynamic type subsumption* one might expect a contravariant condition here.

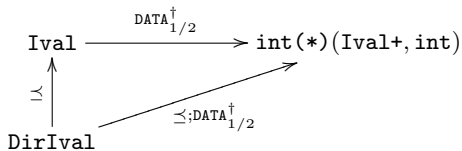
In a *dynamic* setting functions may get passed around. The relevant question is: can function A be called in all contexts where function B can be called? As such when building a dynamic subsumption relation we should treat argument types as assumptions on the calling context and return types as guarantees to the calling context. This would lead to a contravariant definition.

However, in our *static* setting, functions do not get passed around in the same way. The relevant question is: can function A be strengthened to all the signatures to which function B can be strengthened? Static type subsumption (at compile time) is used for a completely different programming intend, as such it should be distinguished from, and possibly used in conjunction with, dynamic type subsumption (at run time).

4.4 Simulation of Strengthenable Function Types

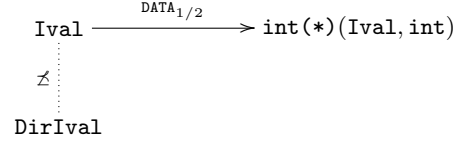
Next we consider how the strengthenable subset of the argument-signature relation $\text{DATA}_{1/2}^{\dagger}$ interplays with the subsumption relation \preceq . For this we consider the example of invoking the DATA operation on an object of type `DirIval`.

In particular we note that $\text{int}(*)(\text{DirIval}+, \text{int})$ is *not* reachable from `DirIval` through the $R_{\text{DATA}_{1/2}}$ argument-signature relation because we have not overloaded DATA to that signature. Instead, we relied on argument strengthening to carry over the definition with type signature $\text{int}(*)(\text{Ival}+, \text{int})$. This is summarized in the following subdiagram of the simulation graph:



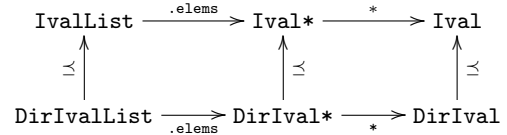
In effect the presence of the $+$ -qualifier prompts us to transitively close the $R_{\text{DATA}_{1/2}^{\dagger}}$ relation over the \preceq relation when selecting a function signature. If we had omitted the $+$ from the definition of

DATA this would mean the first, horizontal arrow would not have been marked with \dagger and hence the second, diagonal arrow would not be present in the diagram, which in turn would mean that this simulation requirement would be violated and `DirIval` would no longer be subsumed by `Ival`:



4.5 Simulation of Pointer Types

Finally we consider how to deal with pointer types. We look at the type subsumption between `IvalList` and `DirIvalList` as introduced in Section 3.2. We see that these types are defined in terms of the types `Ival*` and `DirIval*`. To handle such pointer types we introduce a new type relation R_* such that $(t_*, t) \in R_*$ iff t_* is a pointer-type to type t . This then gives the following subdiagram:



which shows how pointer type subsumption is reduced to type subsumption of the pointed-to types.

4.6 Implementation Issues

We already gave an abstract algorithm for computing the largest valid subsumption relation by iterating a basic fixed point operation. There are several important refinements that can be made to the basic fixed point iteration to make it more efficient.

The first is the computation of $\mathcal{R}_{\preceq}^{\text{init}}$, the syntactical overapproximation of \preceq . It is possible to eliminate many edges in the simulation graph before starting the algorithm proper, just by looking at the surface syntax of the declarations. As an example we mention the comparison of `structs` based on the fields they have available: if a `struct` A misses a field that is present in `struct` B then B for sure does not subsume A. In our current implementation we use a forward definition of syntactical subsumption that uses a bounded depth exploration of `structs` and pointer structures to compare two types for initial subsumption.

In this context it is important to note that we bound the number of aforementioned comparisons by exploiting the pre-order structure using appropriate datastructures. This means a new type can be introduced into the pre-order by traversing the pre-order antichains layer by layer gradually narrowing in on the set of direct parents of the new type. This significantly reduces the number of comparisons that need to be carried out.

A second optimization we apply is to use an edge elimination algorithm based on a waitinglist for the closure procedure rather than a naive fixed point iteration. This approach scales better because it does not require a complete recompute of the simulation relation at each iteration as the naive fixed point computation would.

Upon termination \mathcal{R}_{\preceq} will be a *pre-order*. The typing algorithm we will describe in Section 5 expects a *partial order* on the set of types. Classically this is solved by moving to the set of equivalence classes of types rather than individual types, *or*, alternatively, to mark all the equivalent types as distinct.

For flexibility we give the user the choice to normalize types that share an equivalence class to a single representant *after* hierarchy has been elicited, *or*, alternatively, to mark two or more types as distinct *before* hierarchy is elicited. Because it is not really central

to the current exposition, in the rest of the paper we will assume that \preceq is already a partial order.

4.7 Error Reporting

Whenever the user places an explicit request for type subsumption the compiler will check whether this request is congruent with the inferred type subsumption relation. If it turns out that this is not the case it is important to provide the user with a clear counterexample as to *why* this is not the case.

It is possible to do this by presenting the user, conceptually, with a path through the simulation graph. This path will lead *from* the types that were requested to be in subsumption relation (but were not in actuality), *over* one or more type relation edges, *to* a point in the simulation graph where a clear contradiction is reached.

A contradiction usually means that the requested subsuming type supports an operation that the requested subsumed type does not. Such a path through the type simulation graph can subsequently be turned into a normal C expression, indicating where an expression of the requested subsumed type should be inserted to reach a contradiction and using ellipsis for open terms that are not immediately relevant to the counterexample. As an example of this consider the situation where we request `Ival` to be subsumed by `Iterator`, we can do this using the following syntax:

```
<check Ival subsumed by Iterator>
```

Note that we are not *changing* the subsumption relation in this way. That would not be possible because the subsumption relation is defined mathematically based on the operations that we provide. We are not providing a new operation here, instead, we are merely stating a property about our program which we want to maintain. In this case the property does not hold and the compiler will report the following error:

```
Iterator does not subsume Ival
missing: FIRST( ..., (Ival) );
```

Clearly this gives us enough information to understand why the subsumption does not hold, and also it gives us a starting point if we wanted to fix this situation.

Using rules that turn each type relation that could potentially lead to a contradiction into such a C-like expression term the compiler can output similarly intuitive error messages for pointer types, struct types, function types, etc.

5. Bidirectional Antichain Typing

In this section we present the bidirectional antichain typing algorithm. First, we introduce our definition of *syntax graph*. Next, we introduce some auxiliary definitions concerning *antichains of types*. Finally, we introduce the bidirectional antichain typing algorithm proper.

Definition 2 (Syntax Graphs) Let T be a set of *relevant types* as before. A *type relation* $M \subseteq T \times T$ is *cross-closed* iff for any pair of crossing edges $(t, u'), (t', u) \in M$ such that $t' \preceq t$ and $u' \preceq u$ it holds $(t, u) \in M$. Let $\{M_\gamma\}_{\gamma \in \Gamma}$ be an indexed set of cross-closed *type matching relations*. A *syntax graph* G is defined as a tuple $G = \langle N, E, \gamma \rangle$ where N is a set of *nodes*, $E \subseteq N \times N$ is a set of *edges*, $\gamma : E \rightarrow \Gamma$ is a mapping assigning each syntax edge a type matching relation. We define a *simple typing* $\delta : N \rightarrow T$ as a map that assigns each syntax node a type. We say δ is *valid* iff for all edges $(n, m) \in E$ it holds $(\delta(n), \delta(m)) \in M_{\gamma(n, m)}$. \square

Note that cross-closedness is a symmetric condition: a relation is cross-closed iff its inverse is cross-closed. Further note that if a relation is monotone or antitone it is trivially cross-closed.

A syntax graph is a structure that can be seen as a straightforward generalization of a *syntax tree* where we allow free-form de-

pendencies that transcend the basic syntax tree form. In practice the syntax graph is built over the syntax tree after resolving identifier/declaration dependencies.

The general definition of a syntax graph and type matching relations still allows a lot of freedom in the actual formalization of the particular type system we are interested in. Below we give two particular examples of how this definition is used to encode the typing rules for the MOOT programming layer.

5.1 Typing Struct Field Select Expressions

First we give a basic example of a syntax graph. Consider Figure 6(a). This syntax graph corresponds to the field select expression `i.min` as it was used in the example in Section 3. It consists of two nodes. Node 1 represents the type of the `i` identifier and node 2 represents the type of the `i.min` field. The only edge is labeled with the type matching relation `.min`. We will define this type matching relation as:

$$M_{\text{.min}} = \{\text{Ival} \mapsto \text{int}, \text{DirIval} \mapsto \text{int}\}$$

This type matching relation can be directly transferred from the type simulation graph of Section 4, i.e.: $M_{\text{.min}} = R_{\text{.min}}$. In Figures 6(b) and 6(c) we show two examples of a simple typing that assigns each node a type. Note that only the typing shown in Figure 6(c) is valid.

5.2 Typing Function Call Expressions

As a second example we will show how function call expressions can be represented and typed. Consider Figure 9(a). This syntax graph corresponds to the function call expression `DATA(x, y)` as it was used in the example in Section 3. It consists of four nodes. Node 1 represents the type of the first argument, node 2 represents the type of the second argument, node 3 represents the declared signature type of the function that is being called, and finally node 4 represents the type of the result that is returned by the function.

As can be seen the syntax graph in Figure 9(a) is decorated with three different type matching relations. One for the first argument type, one for the second argument type and one for the return type. These three relations together determine the call matching semantics. From Section 3.8 we recall the notion of *strongest call semantics*. According to this semantics we may only invoke a function if each of the argument types in its declared signature is the best possible match to the actual arguments that are provided. It is possible to enforce this by defining the type matching relation accordingly. For the example this becomes:

$$\begin{aligned} M_{\text{DATA}_{1/2}^{\text{arg}}} &= \\ &\{ \text{Iterable} \mapsto \text{any}(*)(\text{Iterable}, \text{Iterator}), \\ &\quad \text{int} \mapsto \text{int}(*)(\text{int}, \text{int}), \\ &\quad \text{Ival} \mapsto \text{int}(*)(\text{Ival}, \text{int}), \\ &\quad \text{DirIval} \mapsto \text{int}(*)(\text{Ival}, \text{int}) \} \\ M_{\text{DATA}_{2/2}^{\text{arg}}} &= \\ &\{ \text{Iterator} \mapsto \text{any}(*)(\text{Iterable}, \text{Iterator}), \\ &\quad \text{int} \mapsto \text{int}(*)(\text{int}, \text{int}), \\ &\quad \text{int} \mapsto \text{int}(*)(\text{Ival}, \text{int}) \} \\ M_{\text{DATA}_{1/2}^{\text{ret}}} &= \\ &\{ \text{any} \mapsto \text{any}(*)(\text{Iterable}, \text{Iterator}), \\ &\quad \text{int} \mapsto \text{int}(*)(\text{int}, \text{int}), \\ &\quad \text{int} \mapsto \text{int}(*)(\text{Ival}, \text{int}) \} \end{aligned}$$

These relations can easily be computed from the type simulation graph. For example the first type matching relation $M_{\text{DATA}_{1/2}^{\text{arg}}}$ can be

computed in terms of the type relations we introduced in Section 4:

$$M_{\text{DATA}_{1/2}}^{\text{arg}} = R_{\text{DATA}_{1/2}} \cup (\preceq; R_{\text{DATA}_{1/2}}^\dagger)$$

Note that, in general, we must take care to remove from the resulting relation any argument–signature pairs that do not satisfy the strongest call semantics. For the example there are no such pairs. Note that the given relations are monotone by virtue of the type simulation requirement, hence they are also, trivially, cross-closed.

In Figures 9(b), 9(d) and 9(e) we show several examples of a simple typing that assigns each node a type. Note that only 9(e) is a valid typing. Figure 9(c) is *not* a simple typing, as can be seen node 3 receives two different types, we will show how to deal with such ambiguous typings in Section 5.4.

5.3 Type Promotion

Since all the type relations that we gave so far satisfy the stronger requirement of monotonicity, i.e.: for all (t, u') , $(t', u) \in M$ such that $t' \preceq t$ it holds $u' \not\prec u$. The reader may wonder why we then need the freedom offered by the weaker requirement of cross-closedness. As an example of an important type matching relation that is cross-closed but not monotone we mention the standard C type–promotion rules (restricted to `int` and `char`):

$$R_{\text{promote}} = \{\text{char} \mapsto \text{int}, \text{char} \mapsto \text{char}, \text{int} \mapsto \text{int}\}$$

We need a relation such as this one to deal with the standard C promotion rules properly. As can be seen this relation is not monotone as `char` can be promoted to `int` or to itself. For space constraints we simplify the treatment of function calls, meaning we will not apply the promotion rules in the remainder of this paper. We just mention that, in practice, promotion rules can easily be implemented by introducing such an additional relation R_{promote} between the outer argument expression syntax node and the inner function argument syntax node.

5.4 Storing Ambiguous Types as Antichains

The goal of the typing procedure that we will describe in this section will be to arrive at a *simple typing* as introduced in Definition 2. However, *before* this goal is reached, so *during* the typing process, it may occur that we must keep more than one alternative type as the information that is incident on a node from various directions is being processed.

In Figure 9(c) this is illustrated: because the type of the first argument node 1 is not yet fixed (perhaps this information is still being propagated elsewhere in the syntax graph) we have to keep two alternative types for the signature node 3. Despite this ambiguity in function signature, the type of the result can be known none-the-less and is being propagated upward in the syntax graph to node 4.

Typically, type ambiguity would be dealt with by moving to the full powerset lattice of types, i.e. we would annotate the nodes of the syntax graph with *subsets of types* rather than individual concrete types. The downside of this is that the typing process becomes prohibitively expensive to perform because we must keep track of arbitrary subsets of the set of relevant types. For this reason we propose to move to the lattice of *antichains of types* instead.

This has the advantage of providing a useful form of abstraction. Because we are approximating the valid typing from above in the lattice of antichains of types it means we may, at all times, *restrict to the maximal (weakest) types*. In practice this is efficient and the loss in precision turns out to be modest. Overfitting of typings is automatically ruled out: typing ambiguities are always detected. Formally the lattice of antichains of types is defined as follows.

Definition 3 (Antichains) An *antichain of types* $A \subseteq T$ is a set of types that are *pairwise incomparable*, i.e.: for all $t, t' \in A$

it holds neither $t \prec t'$ nor $t' \prec t$. For a given subset of types $U \subseteq T$ with U^* we denote the *downward closure* of U , defined as $U^* = \{t \in T \mid \exists u \in U. t \preceq u\}$, with $\lceil U \rceil$ we denote the restriction of U to *maximal elements* defined as $\lceil U \rceil = \{u \in U \mid \nexists u' \in U. u \prec u'\}$. Note that $U \subseteq T$ is an antichain iff $\lceil U \rceil = U$. With $\mathcal{A}[T]$ we denote the set of antichains of types $\mathcal{A}[T] = \{U \subseteq T \mid \lceil U \rceil = U\}$. We define an ordering \sqsubseteq on antichains such that $A \sqsubseteq B$ iff $\forall a \in A. \exists b \in B. a \preceq b$, we say B subsumes A . Note that $A \sqsubseteq B$ iff $A^* \subseteq B^*$. We define $A \sqcup B = \lceil A \cup B \rceil$. Note that $(A \sqcup B)^* = A^* \cup B^*$. Finally note that $\langle \mathcal{A}[T], \sqsubseteq, \sqcup \rangle$ forms a complete lattice. \square

In Figure 5 we show an example of an antichain of function types together with the associated downward closed set of types. We consider antichains an efficient, compact symbolic representation of the downward closed set of types. For example, the antichain annotating node 3 in Figure 9(c) should be interpreted as such: the node must be typed with some type from the downward closed set of types spanned by its antichain as shown in Figure 5.

If for example, we would now introduce the additional information that node 1 is of type `int` then the antichain for node 3 would converge further to the singleton $\{\text{int}(*)(\text{int}, \text{int})\}$. This is exactly the goal of the typing procedure: to arrive at a singleton antichain so that the node has a single well defined type. The following definition makes this precise.

Definition 4 (Strengthenable Typing) For a given syntax graph G we define a *strengthenable typing* as a map $\Delta : N \rightarrow \mathcal{A}[T]$ assigning each node an antichain of types. If for some $n \in N$ it holds $|\Delta(n)| > 1$ we say the strengthenable typing is *ambiguous*. If for some $n \in N$ it holds $\Delta(n) = \emptyset$ we say the typing is *inconsistent*. If Δ is consistent and non-ambiguous it may be turned into a simple typing δ such that $\delta(n) = t$ iff $\Delta(n) = \{t\}$, in this case we say Δ is *valid* iff δ is valid. We define an ordering \sqsubseteq on typings such that $\Delta \sqsubseteq \Delta'$ iff for all $n \in N$ it holds $\Delta(n) \sqsubseteq \Delta'(n)$. \square

Note that a strengthenable typing Δ can be seen as an element in the product lattice $\mathcal{A}[T]^N$. This product lattice will be the main lattice on which we will define the bidirectional antichain typing algorithm in Section 5.5

5.5 Bidirectional Antichain Typing Algorithm

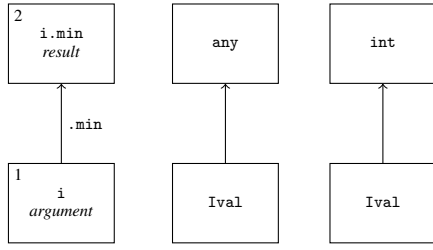
In this section we present the bidirectional antichain typing algorithm proper. The algorithm will work by approximating, from above, a valid typing in the lattice of strengthenable typings. Starting from some initial typing (which should reflect the type declarations and type-casts provided by the user) we descend in the lattice

Algorithm 1 Type the given abstract syntax graph.

Require: A syntax graph: $G = \langle N, E, \gamma \rangle$, an initial typing: Δ

```

1: Waiting  $\leftarrow E$ 
2: while Waiting  $\neq \emptyset$  do
3:    $(n, m) \leftarrow \text{selectFrom}(\text{Waiting})$ 
4:    $(A, B) \leftarrow F_{\gamma(n, m)}(\Delta(n), \Delta(m))$ 
5:   if  $A \neq \Delta(n)$  then
6:      $\Delta(n) \leftarrow A$ 
7:     Waiting = Waiting  $\cup \{(n', m') \in E \mid n \in (n', m')\}$ 
8:   end if
9:   if  $B \neq \Delta(m)$  then
10:     $\Delta(m) \leftarrow B$ 
11:    Waiting = Waiting  $\cup \{(n', m') \in E \mid m \in (n', m')\}$ 
12:   end if
13:   Waiting  $\leftarrow \text{Waiting} \setminus \{(n, m)\}$ 
14: end while
```



(a) Syntax Graph (b) Initial Typing (c) Next Typing

Figure 6. Example of type information flowing forward.

of strengthenable typings by propagating type information along the edges of the syntax graph through application of a bidirectional flow function. The following definition makes this precise.

Definition 5 (Antichain Typing Algorithm) For each $\gamma \in \Gamma$ we define a *bidirectional flow function* $F_\gamma : \mathcal{A}[T]^2 \rightarrow \mathcal{A}[T]^2$ such that for all $A, B \in \mathcal{A}[T]$ it holds:

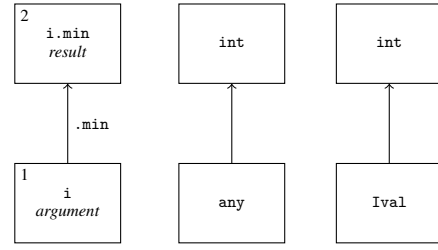
$$F_\gamma(A, B) = ([\{a \in A^* \mid \exists b \in B^*. (a, b) \in M_\gamma\}], [\{b \in B^* \mid \exists a \in A^*. (a, b) \in M_\gamma\}])$$

i.e. the new antichains consist of the weakest types from the left and right downward closed sets for which there exists at least one underlying type relation edge into the opposing downward closed set. \square

In Section 5.7 we show how to compute the bidirectional flow function efficiently, avoiding iteration over the Cartesian product of the downward closed sets. There we also explain why the function cannot easily be split in two separate flow functions. The *Bidirectional Antichain Typing Algorithm* Algorithm 1 shows the bidirectional flow function being applied. Because the flow function is bidirectional the algorithm can infer the type of the argument expressions of some operator expression based on the required result type of the operator expression. This allows us to deal with the situation where the type of a declared identifier must be inferred from the required result type of the operations in which it takes part. To illustrate the algorithm we first look at three different applications of its basic step: the bidirectional flow function.

In Figure 6 we show how type information can be propagated in a forward direction. First we show, in Figure 6(a), the syntax graph for the expression `i.min`. Next we show, in Figure 6(b), an initial typing that assigns the `Ival` type to the argument node of the field select expression and the `any` type to the result node. Finally we show, in Figure 6(c) the result of applying F_{min} once on the initial typing. As can be seen the net effect of the flow function is that the type of the field (`int`) is derived from the type of the `struct` (`Ival`). This constitutes one particular example where type flows in a *forward direction* over the edges of the syntax graph.

In Figure 7 we show how type information can be propagated in a backward direction. First we show, in Figure 7(a), the syntax graph for the expression `i.min`. Next we show, in Figure 7(b), an initial typing that assigns the `int` type to the result node of the field select expression and the `any` type to the argument node. Finally we show, in Figure 7(c) the result of applying F_{min} once on the initial typing. As can be seen the net effect of the flow function is that the type of the `struct` (`Ival`) is derived from the type of the field (`int`). This constitutes one particular example where type flows in a *backward direction* over the edges of the syntax graph.



(a) Syntax Graph (b) Initial Typing (c) Next Typing

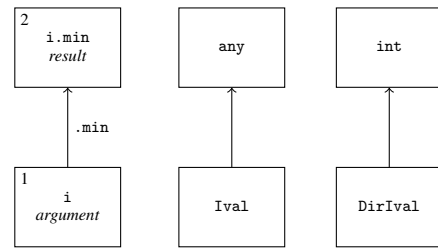
Figure 7. Example of type information flowing backward.

In Figure 8 we show how type information can be propagated in a bidirectional fashion. First we show, in Figure 8(a), the syntax graph for the expression `i.delta`. Next we show, in Figure 8(b), an initial typing that assigns the `any` type to the result node of the field select expression and the `Ival` type to the argument node. Finally we show, in Figure 8(c) the result of applying F_{delta} once on the initial typing. As can be seen the net effect of the flow function is that the type of the `struct` (`DirIval`) and the type of the field (`int`) are derived simultaneously. This constitutes one particular example where types flow in *both directions* over the edges of the syntax graph.

Finally, in Figure 9 we show two examples of how the bidirectional antichain typing algorithm solves typing constraints over a more complex syntax graph by repeatedly applying the bidirectional flow function.

In Figure 9(a) we show the syntax graph of the function call expression `DATA(x,y)` from the example in Section 3. Next we show, in Figure 9(b), an initial typing that assigns `int` to the second argument node and an uninformed type to all the other nodes, including the signature node. Next we show, in Figure 9(b) what is the result of running algorithm 1 on this fragment of the syntax graph. In other words: we repeatedly apply the bidirectional flow function until nothing changes anymore. As can be seen the net the result node receives a singleton type `int` but the signature node and the left argument node are still ambiguous because there is not enough information to determine a singleton type for these nodes. In absence of other information we will have to reject the program with an ambiguous typing error in this case.

In Figure 9(d) we show an initial typing that assigns `DirIval` to the first argument node and an uninformed type to all the other nodes. Next we show, in Figure 9(e) what is the result of running algorithm 1 on this fragment of the syntax graph. As can be seen *all*



(a) Syntax Graph (b) Initial Typing (c) Next Typing

Figure 8. Example of type information flowing both ways.

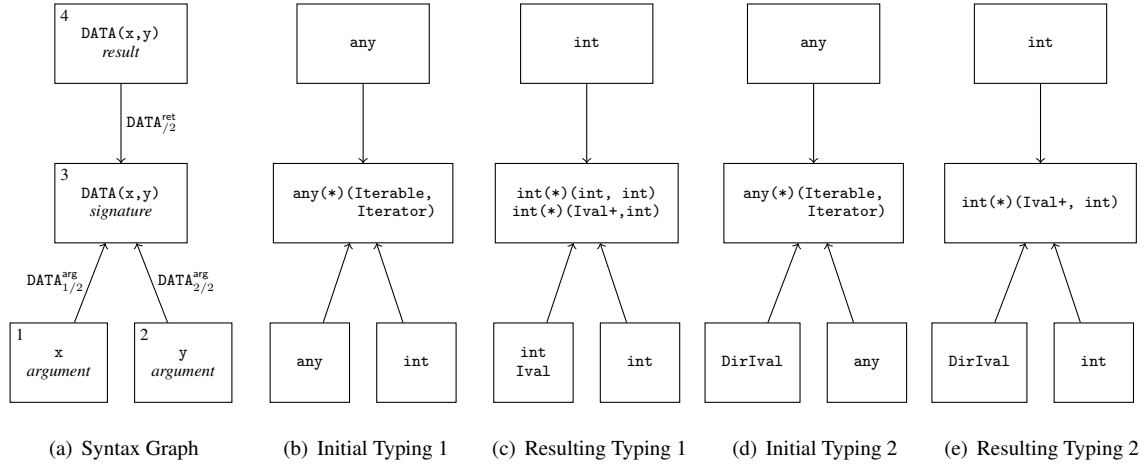


Figure 9. Two examples of the bidirectional antichain typing algorithm when run to completion for two different initial strengthenable typings.

the nodes receive a singleton type. In this case the typing converged and we know which function body to call. The signature node 3 determines the actual function that should be invoked, and the surrounding parameter and return type nodes 1, 2 and 4 determine the actual parameters and return type of the function.

If this is the first time we encounter a call to DATA with these actual parameter types and return type we start the typing procedure on a fresh copy of the syntax graph for the function body of DATA strengthened using the actual argument and return types from nodes 1, 2 and 4 respectively. If, in the process we find a strengthening of one of the arguments and return type of DATA we may propagate this information back to the nodes 1, 2 and 4 respectively. This is a convenient way to handle both the *inter* as well as the *intra*-procedural type flow in a uniform way. Without the need to explicitly deal with higher order type-parameters during the typing process. Note that this procedure is still guaranteed to terminate because the number of types and the number of function definitions is finite, therefore also the number of possible function strengthenings is finite.

We keep the various typings of each function in a spanning tree. This approach allows us to give a lot of context information when typing errors occur. For example, a type error in DATA:

```
type error after call sequence:
1: int main(int, char**)
2: void print( DirIval, char* )
3: void print( DirIval )
4: int DATA( DirIval, int )
...
```

Note that this is not a “real” call-stack as recursive calls are collapsed over their actual signature.

5.6 Correctness

The correctness of Algorithm 1 is ensured by the following two theorems.

Theorem 1 (Soundness) *After termination of Algorithm 1, if Δ is consistent and non-ambiguous then Δ is valid.* \square

Proof. Assume closure is reached and Δ is unambiguous and consistent so that we may define $\delta(n) = t$ iff $\Delta(n) = \{t\}$. We prove that δ is valid. *W.l.o.g. consider some edge $(n, m) \in E$*

and let $\delta(n) = t$ and $\delta(m) = u$ and $F = F_\gamma(n, m)$ and $M = M_\gamma(n, m)$. Because closure was reached it must hold $F(\{t\}, \{u\}) = (\{t\}, \{u\})$ by definition of F this implies that there must exist $t' \preceq t$ such that $(t', u) \in M$ and there must exist $u' \preceq u$ such that $(t, u') \in M$. By cross-closedness this implies that $(t, u) \in M$ as required. \square

Theorem 2 (Termination) *Algorithm 1 will always terminate.* \square

Proof. First observe that the bidirectional flow function F_γ is monotone over the lattice of typings. Next note that the lattice $\mathcal{A}[T]^N$ of typings is finite, so we can go down only a finite number of times before the waitinglist becomes empty. \square

5.7 Efficient Implementation of the Flow Function

In Definition 5 we introduced the bidirectional flow function. However, this definition depends on the Cartesian product between the downward closed sets A^+ and B^+ of types.

This means that a naive, direct implementation of the flow function would need to iterate over this product in order to compute the resulting pair of antichains. This defeats the purpose of using antichains as a symbolic representation for their downward closed sets. Especially for generic types the downward closed sets can become quite large. As a particular example, the downward closed set of the top antichain $\{\text{any}\}^+$ contains *all* the relevant types T .

Therefore, in this section, we give an alternative, equivalent formulation of the bidirectional flow function which avoids this explicit iteration over the downward closed sets. The following definition makes this precise.

Definition 6 (Symbolic Flow Functions) We lift the standard join \sqcup on antichains of types as given in Definition 3 to *pairs* of antichains such that $(A, B) \sqcup (A', B') = (A \sqcup A', B \sqcup B')$. We now define the symbolic bidirectional flow function \mathcal{F}_γ for a given type matching relation $\gamma \in \Gamma$ such that

$$\mathcal{F}_\gamma(A, B) = \bigsqcup_{\underline{a} \in A, \underline{b} \in B} F_\gamma(\{\underline{a}\}, \{\underline{b}\})$$

This formulation avoids an explicit enumeration over the Cartesian product of the downward closed sets A^+ and B^+ , and instead expresses the flow function for arbitrary antichains in terms of the

Cartesian product of A and B directly (note that A and B are usually much smaller than A^* and B^* , and, typically even singletons). The inner call is still an application to the old definition of F_γ but this is a very specific case, namely: F_γ applied only to singleton antichains. In effect, the fact that these antichains are singleton means they now represent *concrete* types.

The following theorem ensures the soundness of this optimization.

Theorem 3 *It holds $\mathcal{F}_\gamma(A, B) = F_\gamma(A, B)$.* \square

Proof. Starting from the definition of F_γ :

$$\begin{aligned} F_\gamma(A, B) &= \\ & \quad (\lceil \{a \in A^* \mid \exists b \in B^*. (a, b) \in M_\gamma\} \rceil, \\ & \quad \lceil \{b \in B^* \mid \exists a \in A^*. (a, b) \in M_\gamma\} \rceil) \\ F_\gamma(A, B) &= \\ & \quad (\sqcup_{\underline{a} \in A, \underline{b} \in B} \{a \in \{\underline{a}\}^+ \mid \exists b \in \{\underline{b}\}^*. (a, b) \in M_\gamma\}, \\ & \quad \sqcup_{\underline{a} \in A, \underline{b} \in B} \{b \in \{\underline{b}\}^+ \mid \exists a \in \{\underline{a}\}^*. (a, b) \in M_\gamma\}) \\ F_\gamma(A, B) &= \\ & \quad (\sqcup_{\underline{a} \in A, \underline{b} \in B} \lceil \{a \in \{\underline{a}\}^+ \mid \exists b \in \{\underline{b}\}^*. (a, b) \in M_\gamma\} \rceil, \\ & \quad \sqcup_{\underline{a} \in A, \underline{b} \in B} \lceil \{b \in \{\underline{b}\}^+ \mid \exists a \in \{\underline{a}\}^*. (a, b) \in M_\gamma\} \rceil) \\ F_\gamma(A, B) &= \\ & \quad \sqcup_{\underline{a} \in A, \underline{b} \in B} F(\{\underline{a}\}, \{\underline{b}\}) = \mathcal{F}_\gamma(A, B) \end{aligned}$$

\square

The utility of this definition is that it reduces the flow function for arbitrary antichains to a finite antichain join over F_γ applied to singleton flow pairs. These singleton flow pairs can easily be pre-computed for each relevant pair. Note that, in particular, $F_\gamma(\{t\}, \{u\}) = (\{t\}, \{u\})$ iff $(t, u) \in M_\gamma$. From this observation it is not hard to develop a closure procedure that efficiently pre-computes F_γ for all the relevant singleton pairs that lead to non-empty resulting pairs.

In practice we will pre-compute F_γ for all relevant singleton pairs and place the results in a sparse lookup table for dynamic programming. Whenever the algorithm requests an evaluation of F_γ on some pair (A, B) of non-singleton antichains that has not been seen before we use definition 6 to reduce the result to a finite join over F_γ applied to singleton pairs (which are guaranteed to be in the lookup table). Once the result is known we add it to the lookup table so that the next time we evaluate $F_\gamma(A, B)$ this will come at the cost of a single lookup. Because the bulk of the evaluations to F_γ are highly repetitive this greatly speeds up the typing process.

6. Conclusion

In this paper we have presented a new approach to static typing of generic container types written in an object oriented style. Our contribution is twofold. First we have shown how to use an adapted definition of a simulation pre-order to automatically infer type-hierarchy in a structural type system that supports argument strengthening. Second we have shown how to use an antichain based representation for types to efficiently implement the resulting type system.

The choice to base the present work on the C++ programming language is mainly a pragmatic one. For the type of high performance (scientific) software that formed the impetus for this work C++ still constitutes the de facto standard, also because of the huge amount of legacy code and libraries that are available. Nevertheless we believe the techniques outlined in this paper are more generally applicable, in particular programming layers for other programming languages can be similarly defined.

As future work on MOOT we are planning to include dynamic type subsumption, separate compilation and memory locality into the programming layer.

6.1 Acknowledgments

We would like to thank Jean-François Raskin and Nicolas Maquet for helpful suggestions, comments and inspiring discussions during the preparation of this paper.

References

- [1] A. Aiken, E. L. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. *Lecture Notes in Computer Science*, 1281:47, 1997.
- [2] Alexander Aiken and Edward Wimmers. Soft typing with conditional types. Technical Report RJ 9454 (83075), IBM Research Division, August 1993.
- [3] Gilad Bracha. *Generics in the Java Programming Language*. Sun Microsystems, 2004.
- [4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [5] Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, pages 169–184, 1995.
- [6] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, November 2006.
- [7] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.
- [8] Kaplan and Ullman. A scheme for the automatic inference of variable types. *JACM: Journal of the ACM*, 27, 1980.
- [9] Uday P. Khedker, Dhananjay M. Dhamdhare, and Alan Mycroft. Bidi-directional data flow analysis for type inferencing. *Computer Languages, Systems and Structures*, 29(1–2):15–44, April/July 2003.
- [10] A. G. Middleton. A case for type and form flow analysis. *Comput. J.*, 20(3):238–241, 1977.
- [11] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [12] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the Association for Computing Machinery*, 6(1):1–17, 1963.
- [13] Nathaniel Nystrom, Vijay A. Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 457–474. ACM, 2008.
- [14] N. Oxhoj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP '92)*, volume 615 of *Lecture Notes in Computer Science*. Springer Verlag, June 1992.
- [15] Schonberg, Schwartz, and Sharir. An automatic technique for selection of data representations in SETL programs. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 3, 1981.
- [16] J. T. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, December 1975.
- [17] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, 2000.
- [18] Aaron M. Tenenbaum. Type determination for very high level languages. Master's thesis, New York University, 1974.
- [19] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV 2006*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.